Analysis of Parallel Sorting Algorithms on Heterogeneous Processors with OpenCL

Thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology

in

Computer Science and Engineering

by

Anshu Raina

(Roll No.- 109CS0636)

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela, Odisha, 769 008, India

May 2013

Analysis of Parallel Sorting Algorithms on Heterogeneous Processors with OpenCL

Thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology

in

Computer Science and Engineering

by

Anshu Raina

(Roll No.- 109CS0636)

Supervisor

Prof. Bibhudatta Sahoo

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela, Odisha, 769 008, India

May 2013

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India.

Certificate

This is to certify that the work in the thesis entitled ***Analysis of Parallel Sorting Algorithms on Heterogeneous Processors with OpenCL*** by ***Anshu Raina*** is carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in the department of Computer Science and Engineering, National Institute of Technology Rourkela.

Place: NIT Rourkela
Date: 10 May 2013

Bibhudatta Sahoo
Professor, CSE Department
NIT Rourkela, Odisha

Acknowledgment

It has been a long journey for me when I started this work and now when I am writing this, I am going to take the privilege of thanking those people who assisted me a lot for completing my work .

First of all, I would like to express my sincere thanks to Prof. Bibhudatta Sahoo for his advice during my thesis work. As my supervisor, he has constantly encouraged me to remain focused on achieving my goal. His observations and comments helped me to establish the overall direction of the research and to move forward with investigation in depth. He has helped me greatly and has been a source of knowledge.

I extend my heartfilled gratitude towards the Computer Center of NIT Rourkela especially Mr. Snehasish Parhi and the team for giving me the opportunity to test my codes on various architectures in the High Performance Computing(HPC) lab that is recently established in NIT Rourkela.

I must acknowledge the academic resources that I have got from NIT Rourkela. I would like to thank administrative and technical staff members of the Department who have been kind enough to advise and help in their respective roles.

Last, but not the least, I would like to dedicate this thesis to my parents and my sister, for their love, patience, and understanding.

Anshu Raina

Abstract

The heterogeneous computing platform with the tremendous raw capacity can be easily constructed with the availability of multi-core processors, high capacitive FPGAs and GPUs which can include any number of these computing units. However, challenge faced until now was the lack of a standardized framework under which the computational tasks and data of applications could be managed easily and effectively. In this thesis, such a framework called OpenCL(Open Computing language) is discussed. OpenCL offers a programmer a single programming framework, which can be used to target multiple platforms from different vendors. Moreover, the appropriateness of OpenCL as a single standard for targeting multiple platforms is analyzed by mapping and optimizing various parallel sorting algorithms to different architectures namely Intel Xeon processor E5-2650 and NVIDIA GPU(Tesla M2090). In addition, the comparison of various sorting algorithm techniques such as Parallel Selection Sort, Bitonic Sort and Parallel Radix Sort is made on the mentioned architectures.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Motivation

Problem Statement

Related Work

Our Contribution

Thesis Organization

Chapter 1

Introduction

The easy availability of multi-core CPUs, high capacity FPGAs and GPUs makes possible a heterogeneous platform with tremendous computational capacity. The previous researches [1–3] have shown that each type of processor technology is suited to implement specific types of functions. Thus an application with many compute intensive segments would be benefitted from a heterogeneous platform that constitutes of different processor technologies. However, these platforms cannot be adapted on a large scale due to the daunting task of programming for such heterogeneous platforms. OpenCL provides a common framework that caters to the need for using a heterogeneous platform. There are four different models that describe OpenCL.

1. Platform Model
2. Memory Model
3. Execution Model
4. Programming Model

The platform model describes a host connected to one or more OpenCL Compute Devices which can be a CPU or a GPU. A Compute Device is a combination of one or more Compute Units, which are further divided into Processing Elements, on which the actual processing takes place.

The execution of an OpenCL program can be divided in two parts: host code which runs on the host device and the device code which runs on one or

more Compute Devices. Kernels and memory objects are managed by the host part under a context through command queue.

In memory model, execution model is mapped. The mapping of work-group takes place onto a Compute Unit, whereas a work-item executes on a PE (Processing Element). Work-items executing a kernel have access to different regions of memory. Global memory gives permission of read/write access to all work-items of every work-group. The accesses of global memory might be cached, depending on the capabilities of the Compute Device. Constant memory is a read-only section of the global memory that remains constant during a kernel execution.

Under the OpenCL programming model, computation can be performed in task parallel, data parallel, or a hybrid of these two models. The major focus of the OpenCL programming model is the data parallel model, where each work-item works on a data item implementing SIMD effectively. Different models are discussed in detail in Section 2.1.

Since we are using sorting algorithms as our case study on OpenCL framework, different sorting algorithms have been discussed and their implementation in OpenCL is given in Chapter 4. There is huge amount of work done in parallel sorting algorithms but as far as the work in OpenCl is concerned there has not been much work.

In the remainder of this Chapter, Section 1.1 gives the motivation of this research, Section 1.2 provides details about the problem statement, Section 1.3 gives the related work done on this topic, Section 1.4 summarizes our contributions and Section 1.5 provides the information about the organization of this thesis.

1.1 Motivation

The programming models of CPUs, GPUs, FPGAs, etc are very different from each other. A move towards a heterogeneous platform makes it even more difficult to give a unified programming model that can work for all architectures. Every existing heterogeneous platform defines its own paradigm of programming

and application development process. To even evaluate such a platform, there is always a learning curve for the application developers. The lack of a standardized framework for application developers is a major obstacle for large scale adaptation of such platforms. OpenCL (OpenComputing Language) seems to be a promising framework to address this issue. Since OpenCL is the only common framework which is supported by all GPU vendors, it becomes a sole candidate to provide a unified programming model for heterogeneous platforms that contain GPUs.

OpenCL is a complete framework that constitutes of a programming language, a set of APIs, and the hardware that supports its constructs. An OpenCL framework implementation or simply OpenCL implementation encapsulates a library that implements the OpenCL APIs, a toolchain for compiling the OpenCL language for the target architecture, computational devices which support concepts of OpenCL, and device drivers for communication with the devices if necessary. It is a possibility that one OpenCL implementation supports different types of devices, e.g. the AMD OpenCL implementation supports CPUs and GPUs from AMD.

Due to active support from CPU and GPU vendors for OpenCL, the existing workstations with supported GPUs are becoming heterogeneous platforms for general purpose computing. OpenCL has started becoming the standard framework for CPU+GPU platforms. The motivation of this research work is to investigate the feasibility of OpenCL as the standard framework for developing applications for heterogeneous platforms with CPUs, GPUs. Moreover, the appropriateness of OpenCL as a single standard framework is tested by performing a case study on Parallel Sorting algorithms in which the algorithms are mapped to different architectures like Intel Xeon Architecture (Intel Xeon E5-2650) and NVIDIA GPU(Tesla M2090).

1.2 Problem Statement

OpenCL seems to be a promising architecture for the developers who use a heterogeneous platform, those who do not want their application to be limited to a specific platform, or want to target more than one platform from a single source-code. The main objective of our work is to find the suitability of OpenCL in targeting multiple platforms which is analyzed by mapping and optimizing various parallel sorting algorithms to different architectures like Intel Xeon Processor (Intel Xeon E5-2650) and NVIDIA GPU(Tesla M2090). The time taken by various parallel sorting techniques on the different architectures mentioned above is the metric of performance on the respective architecture. The sorting algorithms include two implementations of Parallel Selection Sort (both by local memory and global memory), Bitonic Sort and Radix Sort. The performance of these algorithms is measured and a comparative study of these sorting techniques is done on the different architectures.

1.3 Related Work

The OpenCL was released in December 2008 and since its official release it has been exposed to many evaluations. Because most of the functionality is common with CUDA programming standard, OpenCL versus CUDA is a frequently occurring topic in the literature [4–7]. In [4] translation of CUDA implementation of Monte Carlo simulation to OpenCL is done , which shows performance differences varying from 13% to 63%, all in the benefit of CUDA. The differences are attributed to compiler optimization capabilities. In [7] the comparison between CUDA and OpenCL is made on the basis of triangular solver (TRSM) and matrix multiplication (GEMM), which resulted in a slight performance advantage in favor of CUDA. The comparison of CUDA and OpenCL performed in this work shows similarity in differences in performance though less extreme, but what makes it distinguish itself from the other comparisons is that it gives a detailed explanation of the causes for the difference in performance. The compiler optimizations are

not the only factor that is investigated, the launch times of kernel are also taken into account. Moreover, the comparison with the recent NVIDIA SDK shows that the differences in performance between CUDA and OpenCL on NVIDIA GPUs are coming down.

As far as portability aspects are concerned, there is a consensus about usefulness of OpenCL as a single language for different architectures. Performance portability of an OpenCL program becomes hard to achieve [5, 8]. For solving the problem of performance portability, many papers give auto-tuning techniques [5,7,8] in which by extensive profiling selected parameters will be tuned for the architecture, sometimes even search heuristics is used to efficiently explore the design space [7].

The various sorting techniques which are used in this thesis to test the suitability of OpenCL as a single standard framework have a lot of work attached to them though not much in OpenCL but there is a large amount of work done on them in other frameworks. An overview of parallel sorting algorithms can be obtained from [9].The comparative evaluation of performance of sorting algorithms is presented in [10].

Earlier implementations of sorting algorithms on GPU hardware were based on Batcher's Bitonic sort [11]. Bitonic sorting algorithm is implemented using stream processing units and Image Stream processors in [12,13]. An improved version of bitonic sorting network as well as odd-even merge sort is described in [14]. A split based radix sort is described in [15] followed by a parallel merge sort. A similar radix sort in [16, 17] is based on histograms. A high performance parallel radix sort and merge sort routines for manycore GPUs taking advantage of the full programmability offered by CUDA is described in [18].

Most of the sorting algorithms mentioned above are based on general purpose programming model such as CUDA and some are based on traditional graphics-based General- Purpose computing on GPUs (GPGPU) programming interfaces, such as OpenGL and DirectX API. As far as OpenCL is concerned, study of various algorithms is still in progress and there is comparatively less work done

on it. A portable OpenCL implementation of the radix sort algorithm is given in [19] where test of radix sort on several GPUs and CPUs is presented. An analysis of parallel and sequential bitonic, odd-even and rank-sort algorithms for different CPU and GPU architectures are presented in [20] where task parallelism using OpenCL is exploited.

1.4 Our Contribution

In this work, the portability of OpenCL framework is established by mapping and optimizing various sorting techniques to different architectures. The main contributions in this thesis are given as:

- The modified version of selection sort is proposed and implemented using both local and global memory on different architectures.
- The bitonic sort is implemented using global memory on different architectures.
- A different approach of radix sort is proposed and implemented on the given architectures.
- A comparison among the various sorting techniques is made on different architectures.

1.5 Thesis Organization

In this chapter, the motivation for implementation of algorithms in OpenCL, the objectives of our work and our contribution is discussed in a nutshell. The organization of the rest of the thesis and a brief outline of the chapters in this thesis are as given below.

In **chapter 2**, we have given an overview of OpenCL and made a comparison between the other existing frameworks and OpenCL.

In **chapter 3**, architectures on which different algorithms have been implemented are discussed.

In **chapter 4**, various sorting algorithms and our proposed implementation of these sorting algorithm techniques in OpenCL is discussed. Moreover, the results of the experiments are given in this chapter followed by the conclusion of the thesis and the additional ideas about the future work.

Chapter 2

OpenCL Overview and Comparison with Other Existing Frameworks

OpenCL Overview

CUDA

Conclusion

Chapter 2

OpenCL Overview and Comparison with Other Existing Frameworks

OpenCL is an open standard targeted to provide software developers a standard framework for easy access to various heterogeneous processing platforms that include highly parallel GPUs, CPUs and other types of processors. The OpenCL standard specifies a programming standard based on C and a set of API. The details about the OpenCL framework can be found in the OpenCL specification [21]. The OpenCL framework can be best described by the four models explained in the next section. Section 2.1.5 gives the execution flow of an OpenCL application. However, before the advent of OpenCL, efforts from the industry have resulted in several programming frameworks like CUDA, OpenMP and the Cell SDK. In this chapter, we limit our discussion to the existing framework CUDA and make a comparison between the OpenCL and CUDA.

2.1 OpenCL Overview

In this section, four models of OpenCL framework have been discussed to describe the core ideas behind OpenCL. Moreover, the execution flow in the OpenCL application is described. The four models that describe the OpenCL framework are as follows :

1. Platform Model
2. Memory Model

3. Execution Model
4. Programming Model

2.1.1 Platform Model

The OpenCL platform model is given in Figure 2.1.A host which is usually a CPU is connected to one or more OpenCL Compute Devices which can be a CPU or a GPU. A Compute Device is a combination of one or more Compute Units, which are further divided into Processing Elements, on which the actual processing takes place.

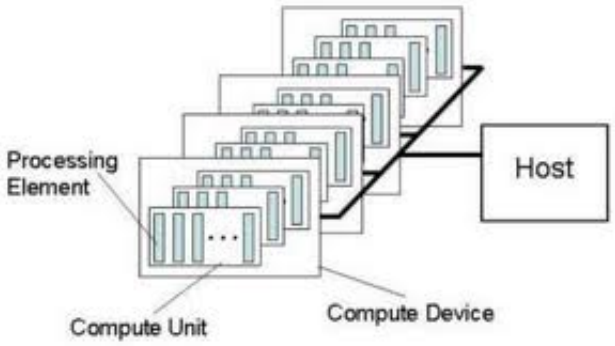

Figure 2.1: OpenCL Platform Model [21]

2.1.2 Execution Model

The execution of an OpenCL program can be divided in two parts: host code which runs on the host device and the device code which runs on one or more Compute Devices. Kernels and memory objects are managed by the host part under a context through command queue.

Context

The context constitutes of all the pieces necessary to use a device for computation purpose. By using the OpenCL API, the host part of the code creates a context object and other objects under it, i.e. kernel object, command queues object, program object and memory objects.

Kernel

The computation that is executed on the processing elements is represented by kernel. We give an example to clarify the kernel concept. Assuming there is an integer array of length 100 and the goal is to add each integer by a constant. Kernel for this problem would only represent addition of one integer by the constant, and the kernel would be instantiated 100 times to solve the complete problem. However, out of consideration for processor utilization and memory access, it is possible to add two integers in the same kernel. If that is the case, the kernel would be instantiated fifty times to solve the complete problem.

Work Items and Work Groups

Kernel execution on a device is defined by an index space, called NDRange. An NDRange is an N-dimensional index space, where N can vary from one to three. The kernel instance is called a work-item. All the work-items execute the same code. However, they usually work on different data and there may be divergence in their execution path through the code. Each work-item is assigned a global ID which is unique throughout the indexed space.

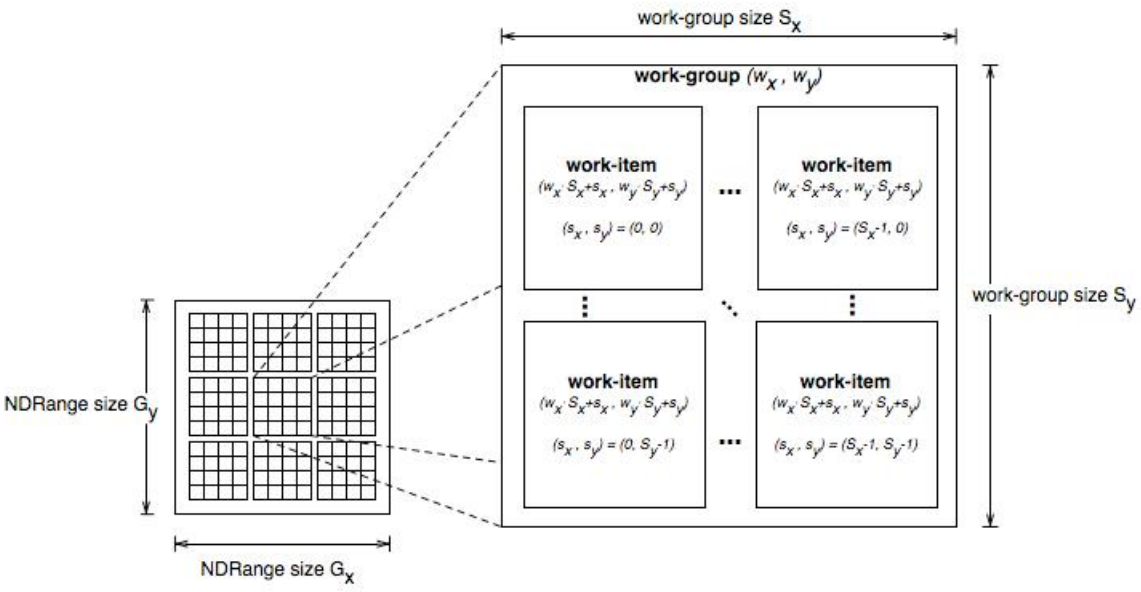

Figure 2.2: OpenCL Execution Model [21]

The equal number of work-items are grouped together to form a work-group. All

the work-groups have same dimensions. The work-item within a work-group has a local ID that is unique across the work-group, and also has access to shared local memory. It is necessary to note here that with proper device support, the total number of work-items may be much greater than the number of processing elements present in a device.

Program and Memory Object

The program object constitutes of the source code and the binary implementation of the kernels. The binary implementation can be generated from the source code during application execution or a pre-compiled binary can be loaded to create the program object. A program object is a library for kernels because one program object may contain multiple kernels. Decides of which kernel to execute during execution is done by application during runtime.

The memory objects are used to transfer data between the host and the device and are visible to both the host and the device. The host creates memory objects, and through the OpenCL API, memory is allocated on the device for the memory objects. The memory model is described in detail in the next section.

Command Queue

The command queue is associated with each device in the context, and memory transfer and kernel execution are coordinated using the command queue. There are three types of commands which can be issued. Memory commands are mostly used to transfer memory between the host and the device. Kernel commands are used to start the execution of kernels on the device. Synchronization commands are used to control the execution order of the commands.

Once the commands have been scheduled on the queue, there are two possible modes of execution. Commands can be executed in-order, meaning the previous command on the queue must have finished execution for the current command to start execution. The other option is that commands execute out-of-order. Here, commands do not wait for previously queued commands to finish execution.

However, synchronization commands can enforce explicit ordering in an out-or-order queue.

2.1.3 Memory Model

The memory model used inside a Compute Device is indicated by Figure 2.3. The execution model as was discussed in 2.1.2 is mapped onto this model. The mapping of work-group takes place onto a Compute Unit, whereas a work-item executes on a PE (Processing Element). Work-items executing a kernel have access to different regions of memory. Global memory gives permission of read/write access to all work-items of every work-group. The accesses of global memory might be cached, depending on the capabilities of the Compute Device. Constant memory is a read-only section of the global memory that remains constant during a kernel execution.

Furthermore, local memory is a region of memory which is only accessible by the work-items inside the same work-group. Depending on the device capability, local memory can be mapped onto the dedicated memory regions of the device or onto the sections of the global memory. Private memory is only visible to the corresponding work-item and is not accessible to other work-items.

2.1.4 Programming Model

Under the OpenCL programming model, computation can be performed in task parallel, data parallel, or a hybrid of these two models. The major focus of the OpenCL programming model is the data parallel model, where each work-item works on a data item implementing SIMD effectively.

Under the OpenCL programming model, computation can be performed in task parallel, data parallel, or a hybrid of these two models. The major focus of the OpenCL programming model is the data parallel model, where each work-item works on a data item implementing SIMD effectively.

The task parallel model can be realized by enqueing multiple kernel exe-

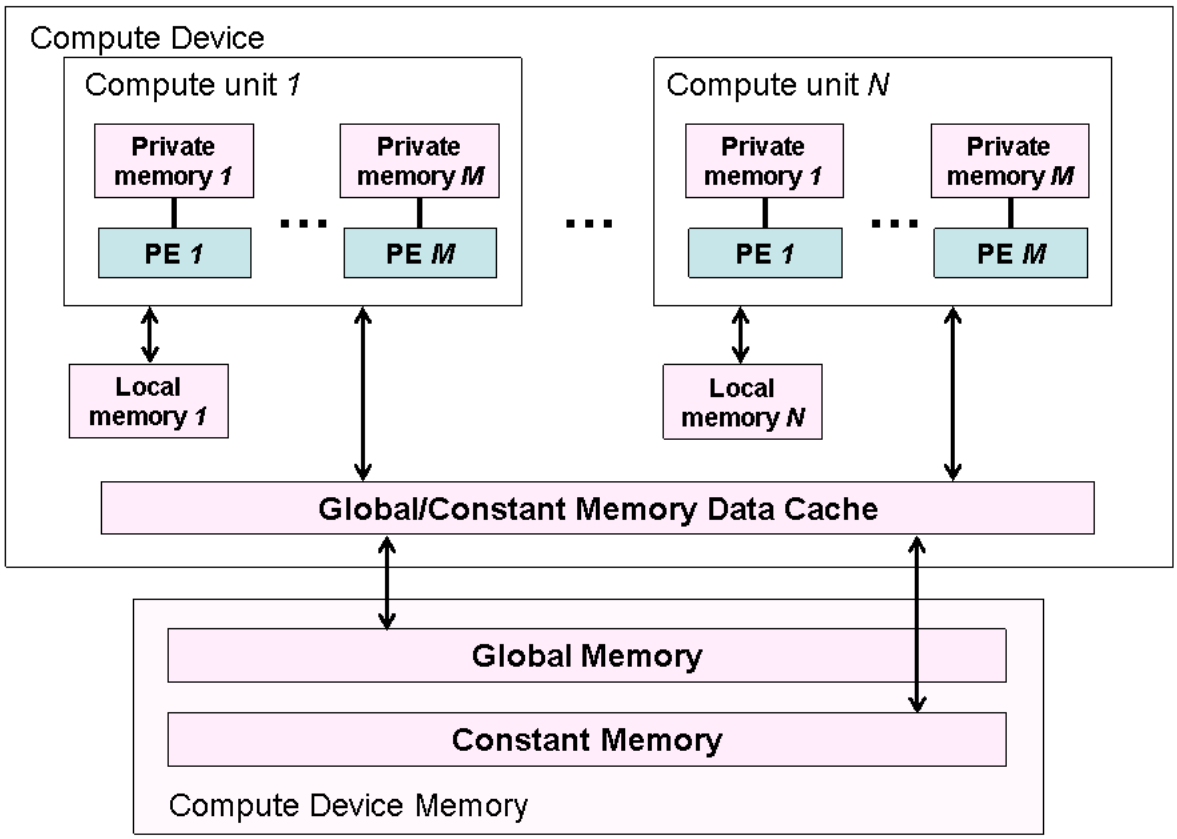

Figure 2.3: OpenCL Memory Model [21]

cution, where only one work-item is created for each kernel. Even though a few GPUs provide support to this model, this is highly inefficient model for the GPUs.

A hybrid model is possible where multiple kernels each with multiple work-items are enqueued for execution at the same time.

2.1.5 Execution Flow in an OpenCL Application

The OpenCL application flow is shown in Figure 2.3. The flow is divided into two sections. A context is created by platform layer based on available platforms, and the runtime layer creates all other necessary objects needed to execute the kernel.

Platform Layer

In an OpenCL application, initially a query is made for available OpenCL platforms. Once the available platform list is assembled, the application chooses the one with the desired device type and a context is created. The possible device types permissible in the OpenCL specification are CL DEVICE TYPE CPU, CL DEVICE TYPE GPU, and CL DEVICE TYPE ACCELERATOR. The desired number of devices from the available devices is added by the context. The devices are made exclusive to the context once added to a context until they are explicitly released from the context.

Runtime Layer

The description of tasks considered to be a part of the run-time layer is given below.

Host and the devices communicate each other using the commands. A command queue is created for each device under the context to issue commands. An optional OpenCL event object can be created, whenever a command is issued. These event objects can be used for explicit synchronization and allow the application to check for the completion of the command.

To allocate memory on the devices, memory objects are created. The application sets the permission to read and/or write to these memory objects from the host when they are created.

By either loading the source code or by the binary implementation of one or more kernels, programs objects are created. The binary representation can be intermediate representation or the device-specific executable. The program objects are then built to generate the device-specific executable once created. The OpenCL implementation decides of the action to be taken in the build stage depending on whether source code, intermediate representation, or an executable was used to create the program object. Writing of the binary implementation to a file is allowed by an OpenCL API that can be used in the later runs of the appli-

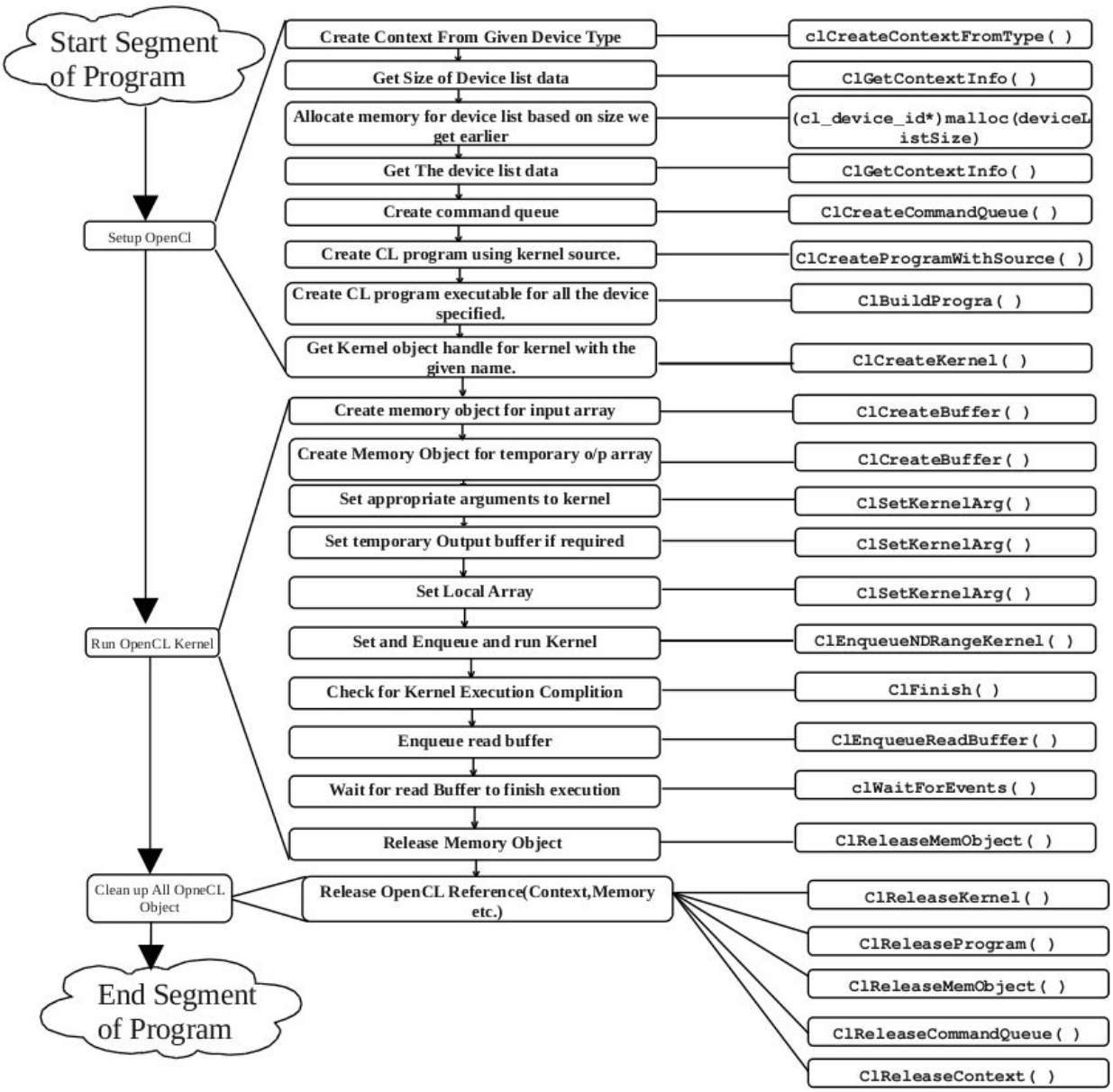

Figure 2.4: OpenCL Application Flow [22]

cation. The format of the output file is not under the OpenCL specification, and the OpenCL implementation decides a format of convenience. The kernel object is created once the executable is built in the program object. One of the functions implemented in the program object is represented by the kernel object.

The input data is transferred to the device memory by issuing memory copy commands against the associated memory objects before executing the kernel. The memory transfer can either be blocking where once the memory transfer

is complete, the control is returned to the application or non-blocking where control is returned after the memory transfer is scheduled. The events are used for synchronization for a non-blocking transfer. The values of the kernel arguments are set once the input data is transferred and the kernel through the command queue is scheduled for execution. The output memory is transferred to the host from the device once the kernel execution is complete. We can have an iterative process where the same kernel is scheduled to run again. New input data can be transferred to the device, and after kernel execution new output data can be transferred back to the host.

2.2 CUDA

Compute Unied Device Architecture, or CUDA, is the name of NVIDIAs parallel computing platform and programming model. It is a full computing platform with a hardware architecture specication, which is supported by extended versions of programming languages existing, an API and a runtime environment. The CUDA hardware is based on the technology of GPU. The GPU, or graphics processing unit, was coined by NVIDIA in 1999 [23]. Around this time, VGA (video graphics array) controllers were advancing to support acceleration of 2D- and 3D-graphics, and the GPU introduced an integrated processing unit that supported that of a traditional high-end workstation graphics pipeline, hence there was a need for a term. Since then, GPUs have steadily become more general, replacing fixed function logic with programmable functionality [24]. The first uses of GPUs for general purpose computing (GPGPU) were obtained by exploiting graphics programming APIs that interfaced with the hardware driver, such as Microsofts DirectX libraries and the open source OpenGL. This was made possible by the well-defined behavior of the APIs. The disadvantage was that the user needed to have intimate knowledge of the APIs and the ability to express programs in terms of graphics.

To address the interest and issues involved with GPGPU programming, NVIDIA defined the unified device architecture and released CUDA C, a version of standard C with the extensions to support GPU programming. The first capable

device of CUDA, representing CUDA capability v1.0 was the G80 architecture, which was first released in 2006. Since then new CUDA-based architectures have added features resulting in updates of the capability specication of CUDA, followed by support in CUDA C. In this section we will highlight features of the hardware, and describe the concepts of the programming model.

2.2.1 CUDA Programming Model

The programming paradigm given by CUDA has allowed developers to use the power of the scalable parallel processors with relative ease, enabling them to achieve speed ups of several times on a variety of applications. Since the release of CUDA by NVIDIA in 2007, a lot of scalable parallel programs were rapidly developed for a broad range of applications, including sorting, matrix-solvers, searching, physics models and computational chemistry. These applications can scale to hundreds of processor cores and thousands of concurrent threads transparently.

CUDA provides some easily understood abstractions that allow the programmer to focus on algorithmic efficiency and develop scalable parallel applications by expression of parallelism explicitly. It provides three key abstractions–a hierarchy of thread groups, shared memory, and synchronization barrier which provide a clear parallel structure to the conventional C code for one thread of the hierarchy. The abstractions guide the programmer to break the problem into coarse sub-problems that can be solved independently in parallel, and then into the finer pieces that can be solved in parallel cooperatively. The programming model scales to large numbers of processor cores transparently: a compiled CUDA program can execute on any number of processors, and physical processor count needs to be known by run time environment [25, 26].

As was explained before, CUDA can also support heterogeneous computation. The serial part of the applications is run on the CPU, and parallel portions are offloaded to the GPU. The CPU and GPU are treated as separate devices which have their own memory spaces. This configuration also allows simultaneous and overlapped computation on both the CPU and GPU without contention for

memory resources. The indispensable part of the code for CUDA is the kernel program. The kernel is a program which operates on the entire stream of data. The context of a CUDA kernel is simply a C code for one thread of the hierarchy, but execution is in parallel across a set of parallel threads. These threads are arranged into a hierarchy of a grid of thread blocks. A grid is a set of thread blocks that can be independently processed on the device by scheduling blocks for execution on the MP and therefore, they may execute in parallel. A thread block is a collection of concurrent threads that can cooperate among themselves through synchronization barrier (where the threads that are generated by a kernel call must wait to synchronize) and threads of a block can only access the shared memory. The execution of thread block takes place as smaller groups of threads known as "warps" the term originates from weaving. So, individual threads that compose a warp start together at the same program address but they are free to execute and branch independently. The size of warp is 32 threads on Tesla architecture.

Each thread has a unique thread ID threadIdx within its thread block, numbered 0, 1, 2, ..., blockDim1, and each thread block has a unique block ID blockIdx within the grid. CUDA supports thread blocks that contains up to 512 threads. The thread blocks may have one, two, or three dimensions, accessed through .x, .y, and .z index fields. Parallelism is explicitly determined by specifying the dimensions of a grid and its thread blocks while launching a kernel. Each kernel launch creates a grid of blocks that assigns one thread to each element of the vectors and distribution of the threads over the blocks takes place. Each thread computes an element index from its thread and block IDs, and the desired calculation on the corresponding vector elements is performed. The representation of CUDA programming model as given in [25] is represented in Figure 2.5.

CUDA code is generally simple and straightforward to write than writing parallel code for vector operations. But, while developing CUDA programs, it is necessary to understand the ways in which the CUDA model is restricted, largely

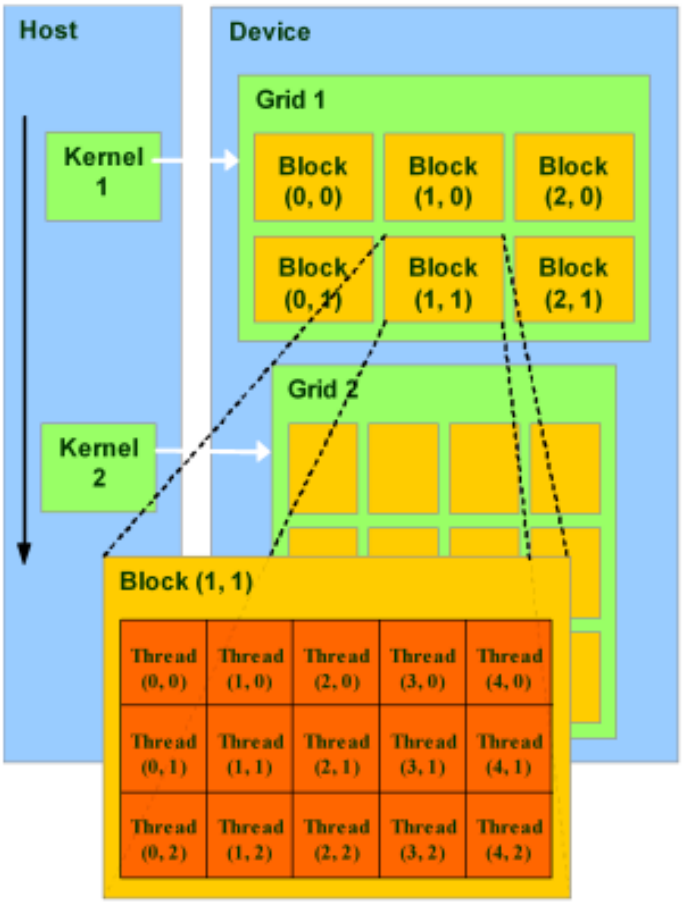

Figure 2.5: Representation of CUDA Programming Model [25]

for the reasons of efficiency. The invocation of kernel in CUDA is asynchronous, so the driver will return control to the application as soon as it has launched the kernel. But, for instance, CUDA functions which perform memory copies are synchronous, and they implicitly wait for all kernels to complete.

CUDA code is generally simple and straightforward to write than writing parallel code for vector operations. But, while developing CUDA programs, it is necessary to understand the ways in which the CUDA model is restricted, largely for the reasons of efficiency. The invocation of kernel in CUDA is asynchronous, so the driver will return control to the application as soon as it has launched the kernel. But, for instance, CUDA functions which perform memory copies are synchronous, and they implicitly wait for all kernels to complete.

During the thread execution, individual threads have access to data that settle in different memory spaces as given by Figure 2.6. Each thread has access to

a private local memory and register file. Each thread block has a shared memory to which all threads of the block have access. Moreover, all threads of different blocks can access same global memory. There are two other read-only memory spaces accessible by all threads : the constant and texture memory spaces as given in Figure 2.6.

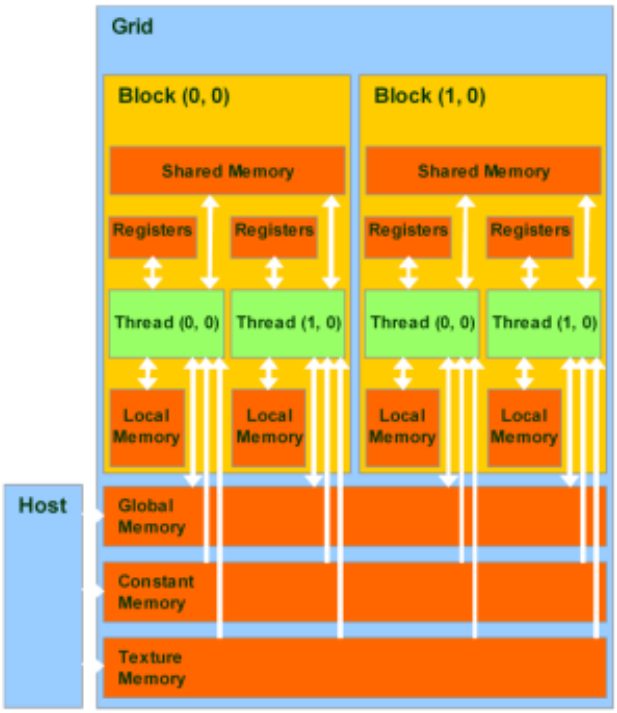

Figure 2.6: Representation of CUDA Threads Blocks mapped on CUDA Memory Model [25]

2.3 Conclusion

In this chapter, the framework of OpenCL and the pre-existing framework CUDA is discussed. While CUDA is a proprietary of NVIDIA, the OpenCL is a open standard managed by Khronos Group. Though CUDA may give slightly more performance on NVIDIA GPUs than OpenCL but the portability aspect of OpenCL and the its property to map any architecture whether its GPU,CPU or any other processor outweighs CUDA and makes it a standard framework which promises a

lot as far as the future of high performance computing is concerned.

Chapter 3

Architectures

NVIDIA Tesla M-Class GPU

Intel Xeon Processor E5 Series Architecture

Conclusion

Chapter 3

Architectures

This chapter discusses about the architectures that are used for the experiments. Currently, OpenCL has been adopted by a growing number of large companies [27]. The devices used for the experiments are listed in Table 3.1. One of them is a Intel Xeon processor (Intel Xeon E5-2650) and the other is a NVIDIA GPU(Tesla M2090).

Table 3.1: OpenCL Target Devices

Vendor	Model(architecture)	SDK(driver version)
NVIDIA	Tesla M2090	OpenCL 1.1 CUDA
Intel	Intel(R) Xeon(R) CPU E5-2650	OpenCL 1.1 (Build 31360.31426)

3.1 NVIDIA TESLA M-CLASS GPU

The Tesla M-class GPU computing Modules are the world's fastest parallel computing processors for high performance computing (HPC) based on the CUDA architecture codenamed Fermi. The high performance of Tesla GPUs makes them ideal for seismic processing, biochemistry simulations, weather and climate modeling, signal processing, computational finance, CAE, CFD and data analytics. Tesla GPUs bring a speedup of 10x in HPC applications. They are based on the Fermi architecture. These GPUs feature up to 665 gigaflops of double precision performance, 1 teraflop of single precision performance, ECC memory error protection, and L1 and L2 caches. The modules of Tesla M-class GPU are integrated into GPU-CPU servers from OEMs. This gives data center IT staff a broad range

of choice in how GPUs will be deployed.

The technical specifications of Tesla M2090 as described in [28] are given in the Table 3.1. Since the architecture of Tesla M2090 is based on Fermi archi-

Table 3.2: Technical Specifications of NVIDIA GPU (Tesla M2090)

Property	Values for Tesla M2090
Peak double precision floating point performance	665 Gigaflops
Peak single precision floating point performance	1331 Gigaflops
CUDA cores	512
Memory size(GDDR5)	6 GigaBytes
Memory bandwidth(ECC off)	177 GBytes/sec
Core Clock	650 MHz
Shader Clock	1300 MHz
Architecture	Fermi
Memory Clock	3.7 GHz GDDR5
Memory Bus Width	384-bit
Transistor Count	3B
TDP	250W

tecture, an abstract overview of the Fermi architecture is presented. The Fermi architecture is a significant step forward in the GPU architecture [23].The device constitutes of a number of Streaming Multiprocessors (SMs). Each SM looks like an SIMD (Single Instruction Multiple Data) processor, which contains 32 cores and 4 Special Function Units (SFU). Every core has a fully pipelined integer Arithmetic Logic Unit (ALU) and Floating Point Unit (FPU). The usage of SFUs is for transcendental instructions, such as sine and cosine. The diagram of streaming multiprocessor(SM) of Fermi as given in [23] is shown in Figure 3.1.

There are several layers in memory hierarchy. The access property of each layer is distinguishable. The memory access latencies are smaller closer to processing elements. The positioning of DRAM is off-chip leading to its largest access latency. Data from the DRAM is cached in L2 cache, which all SMs share. Every SM has its own region of L1 cache along with a Shared Memory region. The amount of L1 cache versus Shared Memory can be configured by the programmer. The major difference between the Shared Memory and the L1 cache is that the former resembles a scratchpad memory, meaning that a programmer has

Figure 3.1: Streaming Multiprocessor of Fermi Architecture [23]

to explicitly read data from and write data into the memory, while the hardware manages the contents of a cache. The memory closest to the processing elements is the Register File, which has the least memory access time.

3.2 Intel Xeon Processor E5 Series Architecture

The Intel Xeon processor E-2650 is based on E5-2600 architecture. In this section, Intel Xeon Processsor E5-2600 architecture is discussed. The architecture is a dynamically scalable micro-architecture that offers up to 8 threads per socket, a maximum of 2 threads per core and up to 20 MB shared cache.The architecture of Intel Xeon E5-2600 is shown in figure 2.2.

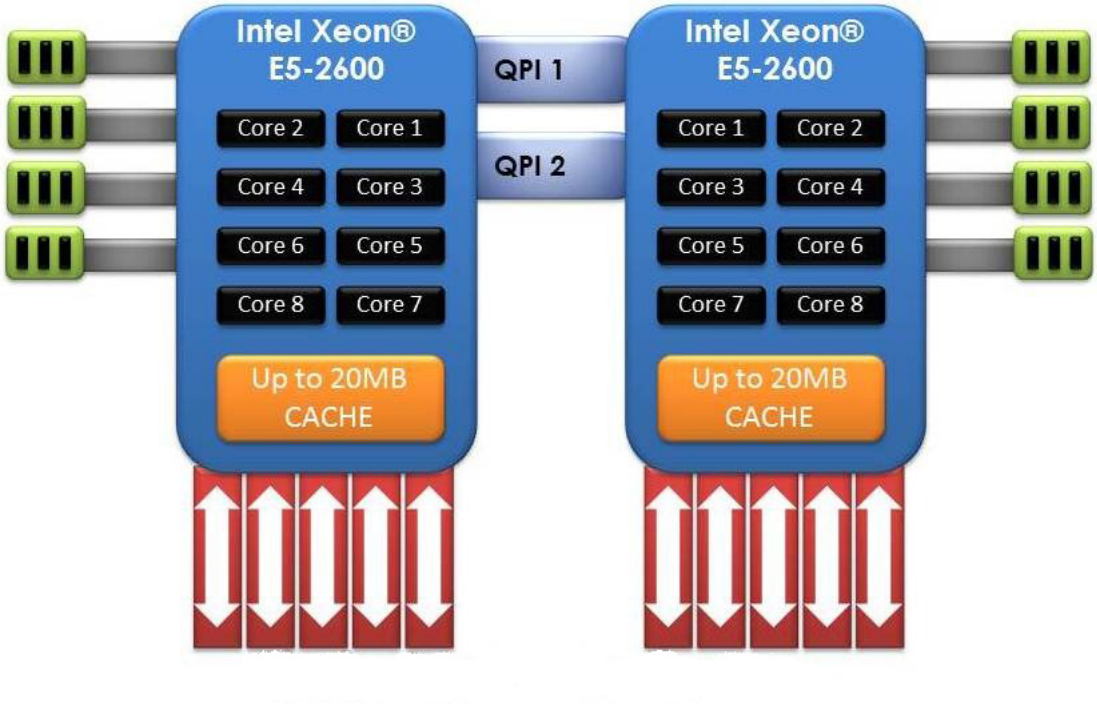

Figure 3.2: Intel Xeon processor E5-2600 Architecture [29]

PCIe 3.0 gives double the bandwidth of PCIe 2.0. The increased I/O bandwidth is required in enterprise deployment with a maximum of 40 lanes per socket. Intel Intelligent Power Technology (IPT) shifts the CPU and memory into lowest available power state automatically, hence reducing energy costs. The hyper-threading technology which the Xeon processor uses, allows thread-level parallelism on each processor, resulting the use of processor in a much efficient way. The optimized Intel Turbo Boost Technology 2.0 tends to maximize the CPU performance during workload spikes by making the CPU operate over TDP. The processor operates

within a closely controlled thermal envelope, controlled by highly-accurate sensors and fuse settings. Intel AES-NI(Advanced Encryption Standard New Instructions) speed ups the encryption and decryption, therefore improving data security. The various characteristics of Intel Xeon E5-2600 as given in [29] are shown in Table 3.2.

Table 3.3: Technical Specifications of Intel Xeon Processor E5-2600

Property	Values for Intel Xeon Processor E5-2600
Sockets	1-2
Number of cores	4, 6 or 8
Frequency	8-core CPUs upto 2.9 GHz
On Die Cache	Upto 20 MB shared L3 Cache
Interconnect Type and Speed	Quick path Interconnect(up to 8.0 GT/s)
Memory Type	DDR3 up to 1600 MT/s
Max Memory Capacity	Upto 768 GB(on boards with 24 DIMMs)
I/O Type	Intel Integrated I/O supporting latest PCIe 3 specification

3.3 Conclusion

In this chapter, various architectures which were used for the experiments were discussed. While Tesla M2090 NVIDIA GPU provides effective speed-up for compute-intensive and parallel applications, the Intel Xeon processor E5-2650 is a well built processor with a number of features that can be harnessed in a parallel application. Therefore, both the architectures can provide appropriate speed up if their architectural aspects are properly utilized.

Chapter 4

Parallel Sorting Algorithms and Implementation in OpenCL

Chapter 4

Parallel Sorting Algorithms and Implementation in OpenCL

This chapter discusses various sorting algorithm techniques and our implementation of these sorting algorithms on the OpenCL framework. The performance of these algorithms is tested on two architectures namely NVIDIA GPU(Tesla M2090) and Intel Xeon E5-2650. Section 4.1 gives a brief overview of Selection Sort and then it subsequently discusses our implementation of parallel selection sort in OpenCL using both global and local memory. Section 4.2 discusses about bitonic sort and its implementation in OpenCL. Section 4.3 discusses about the Radix Sort and subsequently discusses our implementation of parallel radix sort in OpenCL. Section 4.4 gives the results of the experiments performed on different architecture. Section 4.5 summarizes the conclusion of the thesis. Section 4.6 discusses about the future work that can be performed.

4.1 Selection Sort

The sequential selection sort is simple to implement. It is repeated process of finding the largest (or smallest) element and putting the element in its place. Suppose we have to sort elements in the increasing order, we begin by selecting the largest element and moving it to the highest index position or selecting the smallest element and moving it to the lowest index position. We can do this by swapping the element at the highest index and the largest element and in the other case by swapping the element at the lowest index and the smallest element. We then

reduce the size of the array by one element and repeat the process on the smaller (sub) array.

For example, let us consider the following array and we have to sort it in the increasing order:

57, 64, 12, 95, 48

The lowest index here is 0 and the highest index is 4.The largest element in the array is at the index 3. We then swap the element at index 3 with that at index 4. The result is:

57, 64, 12, 48, 95

Now the effective size of array gets reduced to 4, making the highest index in the effective array now 3. The largest element in this effective array (index 0-3) is at index 1, so we swap elements at index 1 and 3:

57, 48, 12, 64, 95

The next two steps give us:

12, 48, 57, 64, 95

12, 48, 57, 64, 95

The last effective array consists of only one element and needs no sorting. The entire array is now sorted. The algorithm for an array, a, with n number of elements can be written down as follows:

Algorithm 1 Sequential Algorithm for Selection Sort

```
for effective_size=limit; effective_size>1; effective_size– do
    find pos_max, the location of the largest element in the effective array: index
    0 to effective_size-1
    swap elements of a at index pos_max and index effective_size-1.
end for
```

In our implementation, we have modified the algorithm of selection sort and we present a parallel version of selection sort that is mapped on OpenCL both in global and local memory. In the modified version of selection sort or called as parallel selection sort, we send the unsorted array onto the device which can be CPU or GPU. Let N be the size of the array. We run N threads or work-items

and each thread or work-item iterates on the entire size of input vector to find the position of that particular element in the array. The position of the particular integer calculated by the thread is updated in the output array. Finally, the output array is sent from the device back to the host. The algorithm for the global memory is as follows:

Algorithm 2 Parallel Selection Sort Kernel Using Global Memory in OpenCL

```
__kernel void SelectionSort(__global int *InValues,__global int *OutValues,__global int *size)
i:=get_global_id(0)
pos:=0
cnt:=0
if i<*size then
  for j:=0;j <*size;j++ do
    if (InValues[j]<InValues[i])or((InValues[j]=InValues[i])and(j<i))
    then
      pos:=pos+1
    end if
  end for
  OutValues[pos]:=InValues[i]
end if
```

We have also implemented Parallel Selection Sort using local memory in which we put the elements up to the workgroup size in the local memory and then use the local array for finding the position of the element in the workgroup taken into consideration. The position of the element found out by respective workgroups are added to get the position of the element in the entire array. The algorithm for the Selection Sort using local memory is given in Algorithm 3.

4.2 Bitonic Sort

A monotonic sequence is the sequence in which the values decrease (or increase) from left-to-right. The sequence a_1,a_2,a_3,a_4...a_{n-1},a_n is monotonically increasing if a_p<a_{p+1} for all p<n.

Algorithm 3 Parallel Selection Sort Kernel using Local Memory in OpenCL

```
__kernel void SelectionSort(__global int *InValues,__global int *OutVal-
ues,__global int *size.__local int *localValues)
i:=get_global_id(0)
l:=get_local_id(0)
wg:=get_local_size(0)
n:=get_global_size(0)
iData:=InValues[i]
pos:=0
for j=0;j<n;j+=wg do
  barrier(CLK_LOCAL_MEM_FENCE)
  aux[l]:=in[j+l]
  barrier(CLK_LOCAL_MEM_FENCE)
  for index:=0;index <wg;index++ do
    if (aux[index] <iData)or((aux[index]=iData)and((j+index) <i)) then
      pos:=pos+1
    end if
  end for
end for
out[pos]:=iData
```

A bitonic sequence is the sequence that monotonically increases (decreases), reaching a single maximum (minimum) and after reaching a maximum (minimum) monotonically decreases (increases). A sequence is also considered bitonic by cyclically shifting the sequence, the sequence becomes bitonic. For example,the following sequences are bitonic.

4 5 7 9 8 6 2 1

5 7 9 8 6 2 1 4

Bitonic sorting uses the property of the bitonic split. A bitonic split is an operation on a bitonic sequence such that if $a_i > a_{i+n/2}$, the two elements are exchanged, $1 \leq i < n$. The operation produces two bitonic sequences A and B such that the all elements in A are less than all the elements in B. By performing bitonic split repeatedly, a bitonic sequence can be converted to a monotonic sequence or a sorted sequence. The bitonic sorter network as given in Wikipedia is shown in Figurte 4.1. An example of sorting a bitonic sequence is given below.

5 8 13 15 10 6 3 1 **Bitonic sequence**

5 6 3 1 10 8 13 15 **First split**

3 1 5 6 10 8 13 15 **Split each half**

1 3 5 6 8 10 13 15 **Split each quarter(exchange)**

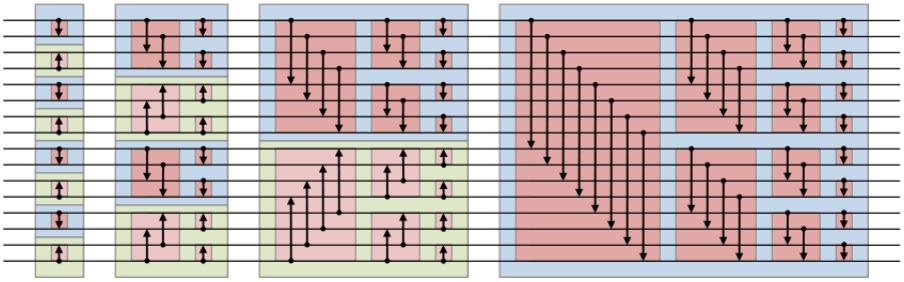

Figure 4.1: Bitonic Sorting network [30]

For sorting an arbitrary sequence, we first turn the sequence into a bitonic sequence and then apply a sequence of bitonic splits as is done above. An example of an arbitrary sequence is given below:

8 13 5 15 1 10 6 3 **Arbitrary sequence**

8 13 15 5 1 10 6 3 **Pairwise exchange alternating order**

8 5 15 13 6 10 1 3 **Split each half (left up, right down)**

5 8 13 15 10 6 3 1 **Pairwise exchange alternating order**

It is noteworthy that the arbitrary sequence of 8 elements has been converted to the bitonic sequence in two steps:

1. The first step consisted of one sub-step i.e. the pairwise exchange.

2. The second step consisted of two sub-steps, the split and the exchange. As we did above, sorting the bitonic sequence took 3 steps. This leads to the following observation. To sort an arbitrary sequence of n-elements, n = 2 ^{k}, takes k steps. Each step has 1,2,3,,k sub-steps.Thus, the total number of substeps are:

$$\sum_{i=1}^{k} k(k+1)/2 = lg(n)(lg(n)+1)/2 = O(lg^2(n)) \tag{4.1}$$

In our implementation of Bitonic Sort in OpenCL, we have implemented bitonic sort in global memory. The direction argument in the algorithm below denotes the direction in which sorting is performed. The value of incr changes in the host

code and we call O(lg^2(n)) kernels and after execution of each kernel we swap the input and output buffers and enqueue a barrier at each invocation.

Algorithm 4 Bitonic Sort Kernel in OpenCL

```
__kernel void BitonicSort(__global int *InValues,__global int *OutValues,__global
int *incr,__global int *direction)
i:=get_global_id(0)
j:=(i)xor(*incr)
iValue:=inValues[i]
jValue:=inValues[j]
if ((jValue<iValue)or(jValue=iValue and j<i))xor(j<i))xor(((*direction) and i) ≠
0) then
   outValues[i]:=jValue
else
   outValues[i]:=iValue
end if
```

4.3 Radix Sort

Radix Sort is a type of sorting algorithm technique that sorts the data consisting of integer keys by distributing each item to a bucket which shares the same significant position or value. After each pass, items are collected in buckets and kept in order and then according to the next significant digit they are again redistributed. Let us suppose that the input elements are 24, 32, 12, 54, 11, 64, 41, 72, 13, 45.
After First Pass : [11,41], [32, 12, 72], [13], [24,54,64], [45]
After Second Pass : [11, 12, 13], [24], [32], [41, 45], [54], [64], [72]
When collected they are in the order : 11, 12, 13, 24, 32, 41, 45, 54, 64, 72
In our OpenCL implementation of radix sort, the histogram is computed looking at the least significant 5 bits of the number. Let N be the input size. For each number, we create N bins of size 2^5 so as to avoid various conflicts like read-write conflict, write-read conflict and write-write conflict because the work-items are executed in parallel. After the histogram formation is over, we collect information from various bins and collect them in an array of size of the 2^5 as we are considering the numbers of only 5 bit. We then perform a prefix sum on the array which

is of size 2^5. Since the prefix sum array is already computed, we can compute the output array by traversing the whole prefix sum array and copying the index from the prefix sum array as many times the difference between the current element and the previous element.

Algorithm 5 Parallel Radix Sort Kernel in OpenCL

Compute the histogram of the input elements.
Perform prefix sum and store the result in prefix-sum array.
Traverse the prefix sum array and copy the index of the prefix sum array to the output array as many times the difference between the current element and the previous element in the prefix sum array.

4.4 Results

This section gives the result of the performance of various sorting algorithms on two architectures namely Intel Xeon Processor (Intel Xeon E5-2650) and NVIDIA GPU (Tesla M2090). The elements to be sorted in all cases were randomly generated 5-bit numbers. The metric of performance is the time taken by the various sorting algorithms. It is to be noted that the time measured here is the run time or the profile time excluding the time for memory allocation, data and memory transfers between the host and the device. The time has been measured through the API clGetEventProfilingInfo. The time taken by various sorting algorithms on NVIDIA GPU (Tesla M2090) is given in Figure 4.2.

The table in Figure 4.2 gives the time taken by various algorithms in seconds. On the basis of the table in the Figure 4.2 a graph is prepared between the input size and the time taken on NVIDIA GPU (Tesla M2090).The graph is represented in Figure 4.3.

Next, results of these algorithms on the Intel Xeon E5-2650 in terms of time taken by these algorithms are given in Figure 4.4.

Again, the table in Figure 4.4 gives the time taken by various algorithms in seconds. On the basis of the table in the Figure 4.4, a graph is prepared between the input size and the time taken on Intel Xeon E5-2650.The graph is represented

Sorting Algorithm	Time (in secs) for n=512	Time(in secs) for n=1024	Time(in secs) for n=2048	Time(in secs) for n=4096	Time(in secs) for n=8192	Time(in secs) for n=16384	Time(in secs) for n=32768	Time(in secs) for n=65536	Time(in secs) for n=131072	Time(in secs) for n=262144	Time(in secs) for n=524288	Time(in secs) for n=1048576
Parallel Selection Sort Global	0.000154	0.000473	0.000914	0.001807	0.003583	0.007163	0.028505	0.113706	0.454653	1.818198	7.277387	29.112523
Parallel Selection Sort Local	0.000133	0.000235	0.000437	0.000839	0.002073	0.007576	0.037599	0.135295	0.510641	1.983154	7.810283	31.004532
Bitonic Sort	0.000457	0.000558	0.000699	0.000847	0.000941	0.001104	0.001467	0.001885	0.002890	0.005597	0.010534	0.021042
Parallel Radix Sort	0.000142	0.000227	0.000341	0.000565	0.001041	0.002006	0.0003929	0.007801	0.015537	0.030851	0.061449	0.122681

Figure 4.2: Table for time taken by the Sorting Algorithms for different Input size on NVIDIA GPU(Tesla M2090)

in Figure 4.5.

4.5 Conclusion of the Thesis

The thesis presents the anaysis of OpenCL to target different architectures by mapping and optimizing various parallel sorting algorithms on different architectures. The conclusions that can be drawn from the thesis are as following :

- The suitability of OpenCL in targeting different architectures is proven as various sorting algorithms are mapped and optimized on different architectures like Intel Xeon E5-2650 and NVIDIA GPU(Tesla M2090). Thus, the portability of the OpenCL framework is established.

- From the tables and the graph, it can be ascertained that bitonic sort provides the best performance when mapped on NVIDIA GPU(Tesla M2090).

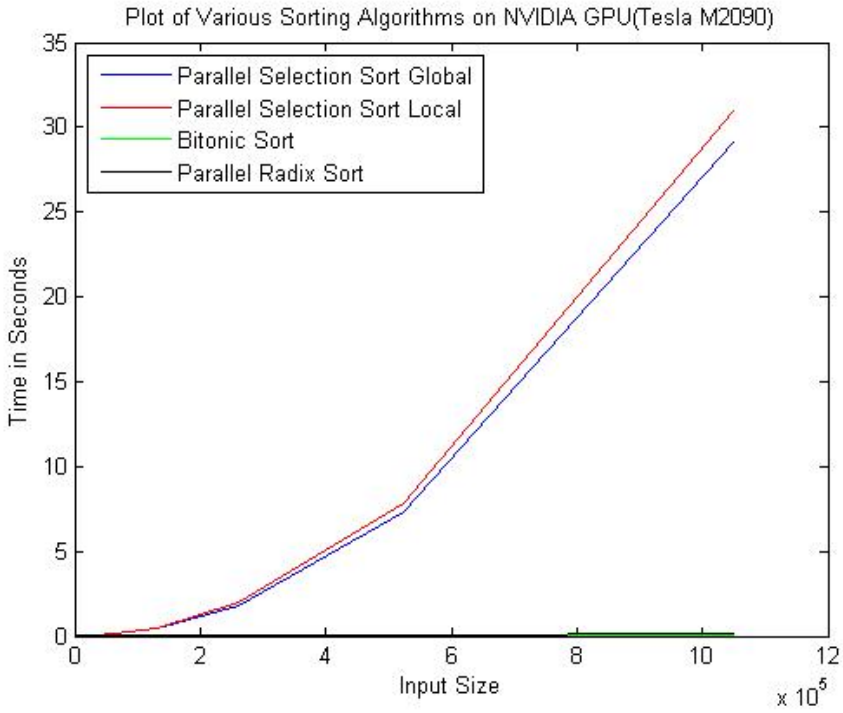

Figure 4.3: Graph for time taken by the Sorting Algorithms versus different Input size on NVIDIA GPU(Tesla M2090)

Sorting Algorithm	**Time in secs for n=512**	**Time in secs for n=1024**	**Time in secs for n=2048**	**Time in secs for n=4096**	**Time in secs for n=8192**	**Time in secs for n=16384**	**Time in secs for n=32768**	**Time in secs for n=65536**	**Time(in secs) for n=131072**	**Time(in secs) for n=262144**	**Time(in secs) for n=524288**	**Time(in secs) for n=1048576**
Parallel Selection Sort Global	0.000599	0.001290	0.004108	0.008369	0.038706	0.138765	0.495688	1.897143	4.948033	16.523541	63.688681	251.949028
Parallel Selection Sort Local	0.001063	0.003325	0.006655	0.015437	0.051790	0.174006	0.662278	2.149684	7.447828	27.342378	106.610431	422.732959
Bitonic Sort	0.001874	0.002397	0.002600	0.003120	0.003428	0.005018	0.007296	0.014773	0.029576	0.054644	0.080841	0.157040
Parallel Radix Sort	0.000386	0.000444	0.000921	0.001140	0.001638	0.002020	0.002744	0.003293	0.006086	0.007963	0.016872	0.032749

Figure 4.4: Table for time taken by the Sorting Algorithms for different Input size on Intel Xeon E5-2650

The results of our findings on NVIDIA GPU (Tesla M2090) are that the bitonic sort is the fastest followed by Parallel Radix Sort, Parallel Selection

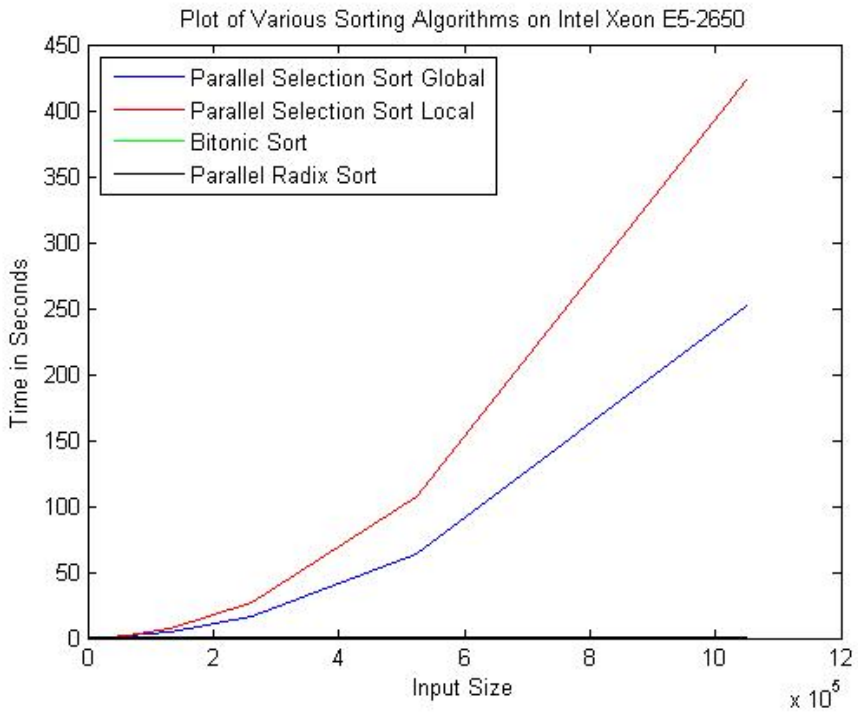

Figure 4.5: Graph for time taken by the Sorting Algorithms versus different Input size on Intel Xeon E5-2650

Local and Parallel Selection Sort Global for input size less than or equal to 8192. However, Parallel Selection Sort Global takes over the Parallel Selection Local after input size increases from 8192. As far as the Intel Xeon E5-2650 is concerned, Parallel Radix sort is the fastest, followed by Bitonic Sort, Parallel Selection Sort Global and Parallel Selection Local.

- It may be noted that that using local memory does not affect the performance of Parallel Selection sort much as it is a kind of a sort which searches the whole array to find the position of the respective element. On the contrary, it may add to the overhead of copying data from global memory to local memory which may mar the performance further.

4.6 Future Work

The following are several areas that can be looked upon further:

- Emphasis can be given on performance portability which can be increased

by some auto-tuning techniques or some other methods.

- To extend the evaluation of OpenCL as a language alternative for current programming standards, a comparison against other native programming standards could be performed. For targeting multi-core CPUs, a comparison against OpenMP or Intel ArBB would be of interest.

- The experiments for measuring the performance of various sorting algorithms need to be performed for more than 5-bit numbers without compensating much on performance.

Bibliography

[1] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating Compute-Intensive applications with GPUs and FPGAs. In *2008 Symposium on Application Specific Processors*, pages 101–107, 2008.

[2] Ben Cope, Peter YK Cheung, Wayne Luk, and Sarah Witt. Have gpus made fpgas redundant in the field of video processing? In *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pages 111–118. IEEE, 2005.

[3] L.W. Howes, P. Price, O. Mencer, O. Beckmann, and O. Pell. Comparing fpgas to graphics accelerators and the playstation 2 using a unified source description. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6. IEEE, 2006.

[4] K. Karimi, N.G. Dickson, and F. Hamze. A performance comparison of cuda and opencl. *Arxiv preprint arXiv:1005.2581*, 2010.

[5] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating performance and portability of opencl programs. In *The Fifth International Workshop on Automatic Performance Tuning*, June 2010.

[6] Matt Harvey. Experiences porting from cuda to opencl. In *Imperial College London CBBL IMIM*, December 2009.

[7] P. Du et al. From cuda to opencl : towards a performance-portable solution for multi-platform gpu programming. In *Electrical Engineering and Computer*

Science Department, University of Tennessee, Technical Report CS-10-656, 2010.

[8] Sean Rul, Hans Vandierendonck, Joris D'Haene, and Koen De Bosschere. An experimental study on performance portability of opencl kernels. In *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*, 2010.

[9] S.G. Akl. *Parallel Sorting Algorithms*. Academic Press, 1985.

[10] Nancy Amato, Ravishankar Iyer, Sharad Sundaresan, and Yan Wu. A comparison of parallel sorting algorithms on different architectures. Technical report, College Station, TX, USA, 1998.

[11] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference, vol. 32*, pages 307–314, 1968.

[12] Timothy J Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 41–50. Eurographics Association, 2003.

[13] A. Greb and G. Zachmann. Gpu-abisort: optimal parallel sorting on stream architectures. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, 2006.

[14] Peter Kipfer and Rüdiger Westermann. Improved gpu sorting. *GPU gems*, 2:733–746, 2005.

[15] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with cuda. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, Aug 2007.

[16] S. Le Grand. Broad-phase collision with cuda. In Hubert Nguyen, editor, *GPU Gems 3*, pages 697–721. Addison Wesley, Jul 2007.

[17] Bingsheng He, Naga K Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 46. ACM, 2007.

[18] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[19] Philippe Helluy. A portable implementation of the radix sort algorithm in opencl. 2011.

[20] B. Montrucchio P. Giaccone F. Gul, O. Usman Khan. Analysis of fast parallel sorting algorithms for gpu architectures. In *in Proceeding FIT '11 Proceedings of the 2011 Frontiers of Information Technology*, 2011.

[21] Khronos group. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[22] Opencl program structure. `http://samritmaity.wordpress.com/2009/11/20/debugging-opencl-program-with-gdb/`.

[23] Nvidia corporation, nvidia's next generation cuda compute architecture: Fermi, 2009. `http://www.nvidia.in/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[24] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, 2009.

[25] NVIDIA Corporation. Nvidia cuda, programming guide version 1.1. July.

[26] NVIDIA Corporation. Nvidia cuda: Compute unified device architecture,reference manual. June.

[27] Khronos group.(2011, june) `http://www.khronos.org/opencl/adopters/`.

[28] Tesla m-class technical specifications. `http://www.nvidia.com/docs/IO/105880/DS-Tesla-M-Class-Aug11.pdf`.

[29] Intel xeon processor e5-2600. `http://www.siliconmechanics.com/files/RomleyInfo.pdf`.

[30] Bitonic sorter. `http://en.wikipedia.org/wiki/Bitonic_sorter`.